Data Encryption: Why the Controversy?

by Jeff Stefan

This article explains the concepts and controversy behind data encryption and provides an encryption program that you can personalize for your own use.

Data encryption has been around in one form or another since mankind started to communicate. When I first think of data encryption, I think of spies decoding secret messages in gray Eastern European alleys, or kids with secret decoder rings deciphering messages on cereal boxes. This outlook, of course, is blatantly naive.

As we evolve toward an information-based economy, more and more of our everyday communications and business affairs are conducted electronically. Millions of financial transactions ranging from Wall Street to the ATM at the corner 7-11 are performed electronically, every day.

This data is exposed to the outside world and needs to be encrypted to preserve privacy and to prevent theft.

The cryptographer's universe was once confined to government intelligence agencies and the dusty halls of a few university mathematics departments. All of this changed with the invention of two things: the personal computer and public key cryptography. The secret world of data encryption has exploded into the public arena and is now in the hands of the everyday computer user.

Some of the encryption methods are easily cracked, and some are considered impossible to break. The once-exclusive academic/government realm of creating and using unbreakable codes is now within the reach of anyone owning a personal computer. This article shows you how these encryption algorithms work.

Data encryption, like any other scientific or mathematical offshoot, has its own peculiar terminology. See the sidebar Cryptographic Terminology for a definition of some of the terms.

Caesar's Alphabet

One of the earliest and simplest encryption techniques, invented by Julius Caesar, is called Caesar's Alphabet. This is a primitive letter substitution cipher. Caesar's Alphabet substitutes D for the letter A, E for the letter B, and so on. This encoding method is easily expressed in C as a two dimensional array, as shown in Figure 1. We can call this array a key table.

The phrase, "we attack at dawn" encoded using Caesar's Alphabet comes out as "zh dwwdfn dw gdzq." The flaws in this encryption method are obvious and many. There is a one-to-one correspondence to the number of letters in a word. Spaces are not hidden, and numbers are not accounted for. The same letter is consistently substituted for the original letter.

In order to attack a cipher like this, start looking for frequently used letters that occur in more than one word. These are usually vowels. The letter d occurs four times, so it probably stands for a vowel .

Look for short words, such as zh and dw. These are usually words like "it," "at," "to," "we," etc. Let's say we pick "dw" to be the word "at" (lucky guess). We assume that d stands for a, and w stands for t. Substituting, the encrypted phrase is now "zh attafn at gazq." This gets us a little closer to the plaintext.

Another way to crack this cipher, after picking "dw" to be the word "at," is to discover that 'd' is three positions away from 'a,' and 'w' is three positions away from 't.' This numerical consistency is a dead giveaway.

The Caesar's Alphabet method of encrypting data is not all bad, though. A more effective approach is to encrypt the entire ASCII character set. Instead of using a fixed substitution offset, a varying offset can be used.

Say that every fourth byte, a new letter will be used to replace the original letter, then every seventh byte, another new letter will be used, and so on. This makes the Caesar's Alphabet method considerably more difficult to crack. This is how the pair of encryption/decryption programs presented in this article work.

The worst problem with this type of encryption technique is that the sending and receiving party must have access to the key table. The key table must be carried to the receiver by hand or transmitted over a secure communications channel. This is called Private Key Encryption, and the key transmission problem is centuries old.

Private Key Encryption

Private Key Encryption is where the sending and receiving parties know the same private key. This is classified as a symmetric key algorithm. One effective and powerful private key algorithm, developed in the 1970s at IBM, is called DES (Data Encryption Standard). DES is so powerful, it was made a standard by the U.S. Government.

DES is used heavily throughout the world for financial transactions. It uses 64-bit blocks of data encrypted with 56-bit keys. DES is not an unbreakable code and, with the sufficient computer hardware and brainpower, DES can be broken.

DES is still quite strong enough to keep random hackers and criminals at bay, and should probably be in use for some time to come. Again, the problem with private key encryption is that both parties must have access to the secret encryption key.

The key must be carried to the receiving site by courier or be transmitted over a secure channel. This is the classic private key encryption problem.

How is this problem eliminated? By making the encryption key available to anyone who wants it. That's the driving principle behind public key encryption.

Public Key Encryption

Public key encryption is brilliant to the core. Instead of using one secret key to encrypt a message, a public key is published that anyone can have access to. If someone wants to send you an encrypted message, they get the public key, encrypt the message, and send it to you. To decrypt the message, you (and only you) have a private key that allows you to extract the original message. This is classified as an asymmetric key algorithm.

It's very difficult, if not impossible, to derive a private key from a public key, and vice versa. Public key encryption was first conceived in 1975 by Whitfield Diffie, a 1960's MIT computer scientist, working with Martin Hellman from

Caesar's Alphabet 1

```
char CaesarAlphabet [2][26] =
    {'a','b','c','d','e','f','g','h','i','j','k','l','m','n','o','p','q','r','s','t','u','v','w','x','y','z',
     'd','e','f','g','h','i','j','k','l','m','n','o','p','q','r','s','t','u','v','w','x','y','z','a','b','c' };
```

Character Encryption Code Fragment 2

```
// Key table data structure
#define NUM_KEYS 4
int KeyTbl[NUM_KEYS] = { 0x04, 0x02, 0x06, 0x01 };


// Encrypt character using the rotating key formula
c = ( c + Key ) % 0xff;

// Increment the character counter
Count += 1;

// Rotate key depending on count
switch ( Count )
{
 case 0x02 :
    Key = KeyTbl[1]; // Key value is 2
    break;
 case 0x03 :
    Key = KeyTbl[2]; // Key value is 6
    break;
 case 0x05 :
    Key = KeyTbl[3]; // Key value is 1
    break;
 case 0x07 :
    Key = KeyTbl[0]; // Key value is 4
    break;
 default:
  break;
}
```

Plaintext

Four score and seven years ago our fathers brought forth on this continent a new nation, conceived in liberty and dedicated to the proposition that all men are created equal. **3**

Encrypted Text

Jswx&tdsvg&goe$wg|ko!}icxy!bks"u{s!jevnkst$ftu{hix$huxui$sp&zijw$"□□dprxktkou$e"tkx!revouo-$gqtifjzif&oo!pmdkxuz$epj&efhmegzfe$xq&zif$$□□ÿ

Stanford University.

In 1977, three MIT scientists — Ron Rivest, Adi Shamirm, and Leonard Alderman — further refined the Diffie-Hellman concept and came up with a set of working public key algorithms. This set of public key algorithms is called RSA, named after the founders. Although patented, the RSA algorithms are widely published.

Here's how the RSA algorithms work. There are two functions, E and D, that encrypt and decrypt messages. E encrypts a message using a public key pair, and D decrypts a message using a private key pair. The public key pair is (e,n), and the private key pair is (d,n). C is the encrypted message, m is the plaintext message, and n is the product of two prime numbers. The number e is the public key, and d is the private key. The trick is to calculate e and d so they work effectively together.

The E and D functions are expressed as:

$E(m) = m^e \bmod n = C$, where C is the encrypted message.

$D(C) = C^d \bmod n = m$, where m is the original message.

These are the steps required to encrypt and decrypt a message:

Step 1: Pick two prime numbers, p and q. For strong encryption, very large prime numbers must be used.

Step 2: Multiply p and q together to produce a new non-prime number n.

Step 3: Choose a random integer d that is relatively prime to (p-1) * (q-1). Again, for strong encryption, this number must be very large. This will be the private key. The number d must satisfy GCD[d,(p-1) * (q-1)] = 1]. This is the mathematical criteria for relative primes. This just means that d and the number produced by (p-1) * (q-1) can only have 1 as a common factor.

Step 4: Find the number e, the public key. e must satisfy the equation:

$$e * d \bmod (p-1) * (q-1) = 1.$$

The sender can now use $E(m) = m^e \bmod n = C$ to encrypt and send a message, and the receiver uses $D(C) = C^d \bmod n = m$ to decrypt the message.

Here's a concrete example:

Step 1: Pick two primes, p and q. We'll pick p = 5 and q = 7 for our example. In the real world, massive primes are used.

Step 2: Our new non-prime number n is 5 x 7 = 35.

Step 3: We need to choose a number d that is a relative prime to (p-1) x (q -1).

(p-1) x (q-1) = (5-1) x (7-1) = 4 x 6 = 24.

A number that is relatively prime to another number means that the only common factor between the two numbers is 1.

That's what the expression GCD[d,(p-1) * (q-1)] = 1] is all about. A good candidate is the number 11, so we'll set d = 11.

Step 4 : Find the number e, the public key. e must satisfy the equation e * d mod (p-1) * (q-1) = 1. We can also use 11 for e. If d = 11, and (p-1) * (q-1) = 24, then 11 * 11 mod 24 = 1.

Our public key e is 11, and our private key d is also 11. The mod number n is 35. We now have all of the numbers required to encrypt a message using the E and D functions. Let's say that we want to encrypt the number 5. We need to run it through the E function,

E(5) = 511 mod 35 = 10. To decrypt the message, we need to run it through the D function, D(10) = 1011 mod 35 = 5, which returns the original value. This encryption method is generally believed to be unbreakable.

PGP, which stands for Pretty Good Privacy, is a popular public key encryption program that is used extensively on personal computers. Pretty Good Privacy is based on the RSA algorithms, and is available from many web sites.

Encryption and the Government

The government is understandably worried about the proliferation of public key encryption. Consider the government's point of view. Their once exclusive and secret cryptographic domain has almost instantly evaporated.

Sure, they can be relatively sure that encrypted messages can be transmitted and received unmolested, but so can everyone else's messages! Secure telephones can be made truly secure, as well as disk files and E-Mail. The private citizen now has access to nearly the same level of encryption technology that the government has.

The government's initial reaction to this dilemma was the production and promotion of the Clipper Chip.

The Clipper Chip works with secure telephones. The initial government proposal was this: you can encrypt your conversations all you want, as long as you use the Clipper Chip and register the encryption keys with the government.

If a government law enforcement agency feels there is sufficient cause to listen in on a conversation, it gets a court order to use the keys held in "escrow" by the Department of Commerce and the Treasury Department.

Using the two keys, along with the data that the Clipper Chip in the secure telephone provides, a government agency can perform a high-tech wiretap on a secure phone. The Clipper Chip proposal was not well received by industry, let alone the general public. See the sidebar "The Clipper Chip" for a further explanation of Clipper Chip technology.

The Encrypt and Decrypt Programs

The programs encrypt.c and decrypt.c are based on the Caesar's Alphabet method of encrypting data, with the addition of a rotating key. The substitution offset (distance from one letter to another) and the substitution letter is changed on a rotating basis. This is

```
/*****************************************************************
* Filename: encrypt.c
* Description: Encrypts a file based on Caesar's Alphabet using
*          the formula c = (c + Key) % 0xff. This is the
*          companion encryption program to decrypt.c.
*
*          Very Important Note: The KeyTbl values in this
*        program MUST match the KeyTbl values in decrypt.c.
*
*  (c) 1997 Jeff Stefan
*****************************************************************/
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>

FILE *Infp;
FILE *Outfp;

#define MAX_FILENAME 45
#define NUM_KEYS     4

int KeyTbl[NUM_KEYS] = { 0x04,0x02,0x06,0x01};
int Key;

void main(void)
{
char InFilename[MAX_FILENAME];
char OutFilename[MAX_FILENAME];
char c;
int i;
long Count=0;

  clrscr();
  /************************
  * Assign first Key value
  ************************/
  Key = KeyTbl[0];

  /*****************
  * open input file
  *****************/
  printf("Enter input filename: ");
  gets(InFilename);
  Infp = fopen(InFilename,"rb");
  if(Infp == NULL)
  {
    printf("Can't open input file..\n");
    exit(0);
  }

  /*******************
  * Open output file
  *******************/
  printf("\nEnter output filename: ");
  gets(OutFilename);
  Outfp = fopen(OutFilename,"wb");
  if(Outfp == NULL)
  {
    printf("Can't open output file..\n");
    exit(0);
  }

  /*************************
  * Encrypt plaintext file
  *************************/
  c = fgetc(Infp);
  while(!feof(Infp))
  {
    /*********************************
    * encrypt character using the
    * rotating key encryption formula.
    *********************************/
    c = (c + Key) % 0xff;

    /*****************************************
    * Rotate Key depending on value of count.
    *****************************************/
    Count+=1;
    switch(Count)
    {
      case 0x02:
        Key = KeyTbl[1]; // Key value is 2
        break;

      case 0x03:
        Key = KeyTbl[2];   // Key value is6 1
        break;

      case 0x05:
        Key = KeyTbl[3];    // Key value is 1
        break;

      case 0x07:
        Key = KeyTbl[0];    // Key value is 4
        Count = 0;
        break;

      default:
        break;
    }
    /************************************************
    * Print the decryted character to screen and file
    ************************************************/
    fputc(c,Outfp);
    printf("[%2X]",c);

    c = fgetc(Infp);
    if(feof(Infp))
      fputc(EOF,Outfp);
  }

  /*******************
  * Close all files
  *******************/
  if(Infp)
    fclose(Infp);
  if(Outfp)
    fclose(Outfp);
}
```

```
/*****************************************************************
* Filename: decrypt.c
* Description: decrypts a file based on Caesar's Alphabet using
*          the formula c = (c - Key) % 0xff. This is the
*          companion decryption program to encrypt.c
*
*          Very Important Note: The KeyTbl values in this
*        program MUST match the KeyTbl values in encrypt.c.
*
* (c) 1997 Jeff Stefan
*****************************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>

FILE *Infp;
FILE *Outfp;

#define MAX_FILENAME 45
#define NUM_KEYS    4

int KeyTbl[NUM_KEYS] = {0x04,0x02,0x06,0x01};
int Key;

void main(void)
{
char InFilename[MAX_FILENAME];
char OutFilename[MAX_FILENAME];
char c;
char x;
int i;
long Count=0;

  clrscr();

  /*********************
  * Assign first Key value
  *********************/
  Key = KeyTbl[0];

  /*****************
  * open input file
  *****************/
  printf("Enter input filename: ");
  gets(InFilename);
  Infp = fopen(InFilename,"rb");
  if(Infp == NULL)
  {
    printf("Can't open input file..\n");
    exit(0);
  }

  /*******************
  * Open output file
  *******************/
  printf("\nEnter output filename: ");
  gets(OutFilename);
  Outfp = fopen(OutFilename,"wb");
  if(Outfp == NULL)
  {
    printf("Can't open output file..\n");
    exit(0);
  }

  /************************
  * Decrypt encrypted file
  ************************/
  while(!feof(Infp))
  {
    c =  fgetc(Infp);
    if(feof(Infp))
      break;
    /*************************************
    * Decrypt character using the inverse
    * rotating key encryption formula.
    *************************************/
    c = (c - Key) % 0xff;

    /*************************************
    * Rotate Key depending on value of count.
    *************************************/
    Count+=1;
    switch(Count)
    {
     case 0x02:
       Key = KeyTbl[1];  // Key value is 2
       break;

     case 0x03:
       Key = KeyTbl[2];  // Key value is 6
       break;

     case 0x05:
       Key = KeyTbl[3];  // Key value is 1
       break;

     case 0x07:
       Key = KeyTbl[0];  // Key value is 4
       Count = 0;
       break;

     default:
       break;
    }
    /*************************************************
    * Print the decryted character to screen and file
    *************************************************/
    printf("[%2X]",c);
    fputc(c,Outfp);
  }
  /*******************
  * Close all files
  *******************/
  if(Infp)
    fclose(Infp);
  if(Outfp)
    fclose(Outfp);
}
```

accomplished by key table array called KeyTbl[NUM_KEYS] and a switch statement that examines the value of a variable named Count. The key table array is declared as int KeyTbl[NUM_KEYS] = {0x04,0x02,0x06,0x01}.

These values are added to the incoming file character, c, to encrypt it in encrypt.c. The values are subtracted from the encrypted file character in decrypt.c to return it to the original character. Only one line of code is required to encrypt the incoming character. This line of code is:

c = (c + Key) % 0xff.

The value of Key added to the value of c, yields a new character.

What happens when we hit the end of the ASCII character set? We can't go beyond that, so that's where the C mod operator %, comes in handy. The last character in the extended ASCII PC character set has a value of 256, which is expressed as 0xff in hexadecimal. Using the % operator removes the 256 and leaves the only value of Key.

For example, if the character you want to encrypt has an ASCII value of 251 (the square root symbol), and the value of Key is 11, then the new value of the character will be 6 (the black club symbol from a deck of cards).

Every character that comes in is counted by the long integer variable Count. The Count value is used to assign a new value to Key. This is done inside a switch statement. The code fragment that encrypts a character is given in Figure 2.

To decrypt an encrypted file, all that needs to be done is to subtract the values that were added to the original character. This simple change in the encryption formula is:

c = (c - Key) % 0xff.

This transforms the encryption formula into a decryption formula. That's the formula that's in decrypt.c.

Figure 3 shows the opening sentence of the Gettysburg Address encrypted with encrypt.c. To return the encrypted text to plaintext, just run the encrypted text through decrypt.c.

Program Customization

You can customize encrypt.c and decrypt.c with your own unique key and count values. You can increase the number of keys in the key table array and use many more count values in the switch statement to increase the encryption complexity. Remember, you must make the key table and switch statement count values exactly the same in both programs. This is a symmetric key algorithm, so both keys must be exactly the same.

The encryption programs work on binary, as well as ASCII files, so you can encrypt executable programs, word processor documents, and spreadsheets, as well as ASCII text files. I use it to encrypt source code and documents I wish to keep private on my machines.

After you encrypt a file, the original file is still left intact. It's up to you to delete it or save it to disk or on some other off-line storage device, if you so desire. The code, as always, is yours to modify and experiment with. I used Turbo C/C++ to compile the source, but any ANSI C compiler should work.

Encryption and the Near Future

The data encryption controversy is not going to go away. On the contrary, it's only going to heat up. A bill called the Secure Public Networks Act of 1997 is currently gathering congressional steam. This bill will mandate government-sponsored key recovery methods and allow law enforcement agencies to confiscate private keys with a search warrant only.

The patents on the RSA algorithms are due to expire in the year 2000. This is when data encryption will intersect free market capitalism, opening a floodgate of new data encryption start-up companies vying for your crypto dollar.

This will make the privacy versus government eavesdropping debate even more intense, and government may try to introduce even more radically restrictive legislation.

It looks as if a showdown is just beyond the horizon. **NV**

Cryptographic Terminology

Cryptograpy	*The science of concealing the true meaning of a message.*
Plaintext	*The original, un-encrypted message.*
Encryption	*The process of encoding plaintext to produce ciphertext.*
Decryption	*The process of un-encoding a message to produce plaintext.*
Cipher	*The encrypted message.*
Key	*A special number, symbol, or set of numbers and symbols that allow an encrypted message to be de-crypted.*
Weak Cryptography	*An encrypted message that is fairly easily decrypted.*
Strong Cryptography	*An encrypted message that is hard or impossible to decrypt.*

The Clipper Chip

The Clipper Chip is a hardware encryption device that utilizes an encryption algorithm called Skipjack. Skipjack was developed by the National Security Agency (NSA). Every Clipper Chip contains a unique encryption key and serial number. The encryption key in the Clipper Chip is derived from two separate keys, which are held escrow by the government. One of the keys is held by NIST, which is part of the Deparment of Commerce, and the other is held at the Automated Systems Division of the Treasury Department.

When two Clipper Chips communicate, a temporary key is first created. The encryption key and the Clipper Chip serial numbers are then encrypted with another key, called the family key. Before the actual data is transfered between Clipper Chips, a Law Enforcement Access Field (LEAF) is exchanged. The data in the LEAF, along with the two keys held in escrow, allow the government to decrypt Clipper Chip messages.